EXHIBIT 11

Autonomous Management of Clustered Server Systems Using JINI

Chul Lee, Seung Ho Lim, Sang Soek Lim, and Kyu Ho Park

Computer Engineering Research Laboratory, EECS
Korea Advnaced Institute of Science and Technology
{*chullee, shlim, sslim*}*@core.kaist.ac.kr and kpark@ee.kaist.ac.kr*

Abstract. A framework for the autonomous management of clustered server systems called LAMA[1] (Large-scale system's Autonomous Management Agent) is proposed in this paper. LAMA is based on agents, which are distributed over the nodes and built on JINI infrastructure. There are two classes of agents: a grand LAMA and ordinary LAMAs. An ordinary LAMA abstracts an individual node and performs node-wide configuration. The grand LAMA is responsible for monitoring and controlling all the ordinary ones. Using the *discovery*, *join*, *lookup*, and *distributed security* operations of JINI, a node can join the clustered system without secure administration. Also, a node's failure can be detected automatically using the *lease* interface of the JINI. Resource reallocation is performed dynamically by a reallocation engine in the grand agent. The reallocation engine gathers the status of remote nodes, predicts resource demands, and executes reallocation by accessing the ordinary agents. The proposed framework is verified on our own clustered internet servers, called the CORE-Web server, for an audio-streaming service. The nodes are dynamically reallocated satisfying the performance requirements.

1 Introduction

Server clustering techniques have been successfully used in building highly available and scalable server systems. While the clustered servers have been enlarging the scale of the service, management has become more complex, as well. It is notoriously diffcult to manage all the machines, disks, and other hardware/software components in the cluster. Such management requires skilled administrators whose roles are very important to maximize the uptime of the cluster system. For instance, configurations of newly installed resources, optimization to get a well-tuned system, and recovery from any failed resources have been performed thoroughly. These days, a self-managing system is promising for its ability to automate the management of a large system, so that the scalable and reliable administration can be achieved.

We have developed our own clustered internet server, called CORE-Web server including a SAN-based shared file system[1], a volume manager[2], L-7 dispatchers[3][4], and admission controllers [5][6]. The complexity of managing the servers led us to develop a framework for autonomous management.

[1] This work is supported by NRL project, Ministry of Science and Technology, Korea

In order to relieve administrator's burden, GUI-based management tools [7] may be utilized. They made it easy to manage a set of clustered nodes with user-friendly interfaces; however, there are still more things to be automated, or to be more robust against failures. Many researchers have studied about self-managing systems, which includes self-configuration, self-optimization, self-healing, and self-protection [8][9]. The autonomous management will be gradually improved to help avoid manual configuration, so that humans would only be needed for physical installation or removal of hardware.

Our goal is also to develop a self-managing CORE-Web server system, while adding some attributes such as flexibility, generality, and security. To achieve our goals, we have built an agent-based infrastructure for the autonomous management, using JINI technology [10], since the JINI infrastructure provides quite useful features to build a distributed system in a secure and flexible manner. Using *discovery*, *join*, *lookup*, and *distributed security* a node can join to the clustered system without administration securely, so that another node can access the newly joined nodes without knowing specific network addresses. Also, a node's failure can be detected using the *lease* interface of the JINI. The leasing enforces each node to renew by a given expiration time, so that the failed nodes can be detected using the expiration of the lease.

We named our autonomous agent as LAMA, which stands for a Large-Scale system's Autonomous Management Agent. LAMAs are implemented on top of JINI infrastructure. Each agent, called LAMA, is spread over each node. During boot-up, the LAMA registers its capabilities to the lookup service (LUS), located in the Grand LAMA, which is the managing LAMA. Other agents might discover the LAMA simply by looking up the LUS, so as to acquire the controls over the nodes.

Next section briefly describes previous works on autonomous management. We then describe LAMA-based autonomous management for dynamic configuration in section 3. Using our CORE-Web server, we built a live audio streaming server, which provides autonomous management.

2 Background

In order to automate the management of a large system, we employed part of off-the-shelf technologies. A newly delivered bare-metal machine can be booted via remote booting like PXES[11] or Etherboot[12]. Also, the node can be booted from a SAN storage. After booting up and running a minimal set of software, we should tune and configure parameters. Individual nodes should be configured for an application service so that the node becomes a trusty component of the service. Many tools are released to enable remote configuration management of a large system, like LCFG[13] and KickStart[14].

While such tools make it easy to manage diverse systems, individual nodes must be able to optimize themselves. AutoTune agents[15] manage the performance of Apache web server by controlling configuration parameters.

The GridWeaver project is aimed to enable autonomous reconfiguration of large infrastructures, according to central policies[16]. Also, many researchers focus on the dy-

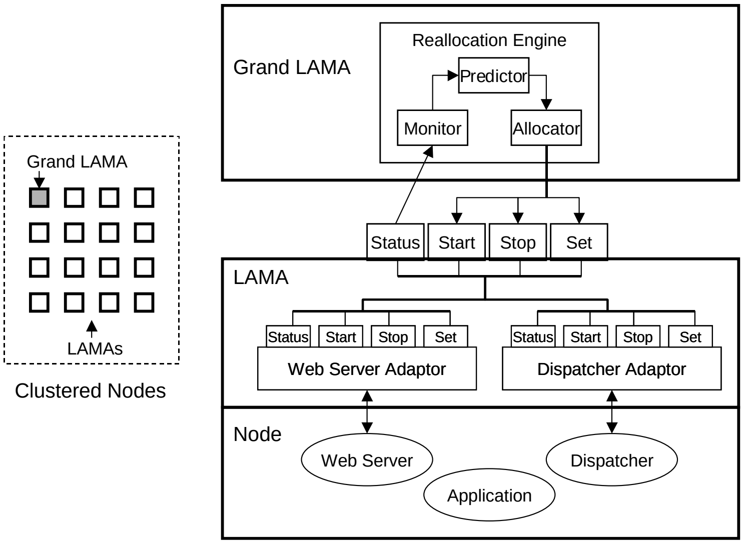


Fig. 1. LAMA architecture

namic reallocation of large infrastructure based on the Service Level Agreement (SLA) like a data center[17][18].

The missing part is a methodology for a systematic development and integration of an autonomous system. Also, important points of autonomous management are run-time optimization and adaptation, which are too hard for humans to perform. Our work considers the issues.

3 LAMA architecture

LAMA is an agent, which is in charge of managing each component in a system. There are two types of LAMAs; ordinary LAMAs and a Grand LAMA. The ordinary LAMAs perform node-wide configuration while residing at individual nodes. The Grand LAMA is responsible for orchestrating all the ordinary ones. As shown in Figure 1, LAMA is a kind of an adaptor that abstracts a node and provides simple control methods to the Grand LAMA. The methods include *Status*, *Start*, *Stop*, and *Set*, through that a reallocation engine in Grand LAMA controls and monitors the nodes. LAMA abstracts detail configurations of specific applications. Inside LAMA, there are several classes of adaptors, and they enable legacy applications to be controlled by the Grand LAMA. For example, a web server adaptor is plugged in to an Apache web server, then the adaptor returns the Apache's status (status), runs up and down the processes (start/stop), or manipulates the Apache's configuration file (set). The adaptor doesn't need any modification of the Apache web server. The Grand LAMA is then able to configure the web server dynamically using four methods mentioned above.

We assumed that the management of a pool containing many nodes would be complex, since the nodes' joins and leaves (failures) might be frequent in a large scale

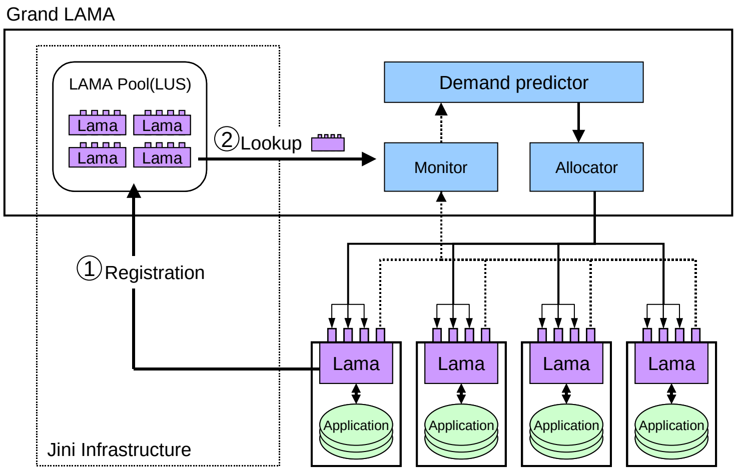


Fig. 2. LAMA pool management

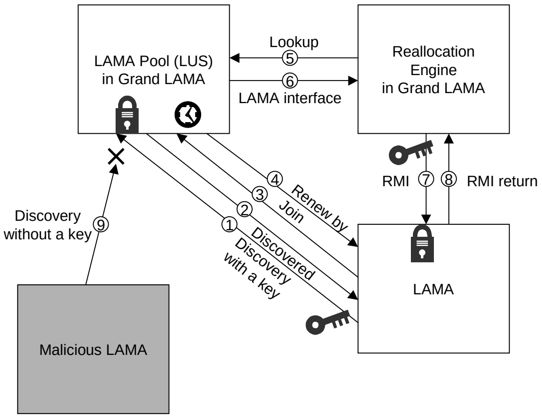
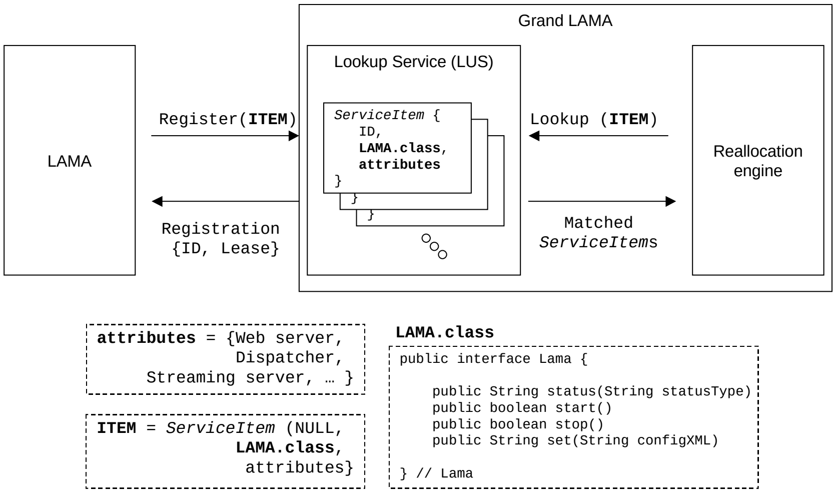


Fig. 3. LAMA in the JINI infrastructure

system. In a traditional way, we would have to manually register all the nodes to the pool by specifying detailed network parameters. Also, we would required numerous manual reconfigurations, upon changing the network configuration.

Our management system uses JINI infrastructure for building a pool without knowing specific network configurations. Figure 2 describes how a LAMA pool is managed. When a LAMA is booted, it discovers the pool (usually known as a Lookup Service or LUS in JINI) and registers its interface automatically. The other modules, like a monitor and an allocator of the Grand LAMA, get the interface to control and monitor the remote nodes. The detailed operations of *discovery*, *join*, *lookup*, and *distributed security* in the JINI infrastructure are described in Figure 3. A LAMA multicasts discovery messages to the network, and the LAMA pool (LUS) in the Grand LAMA responds with a



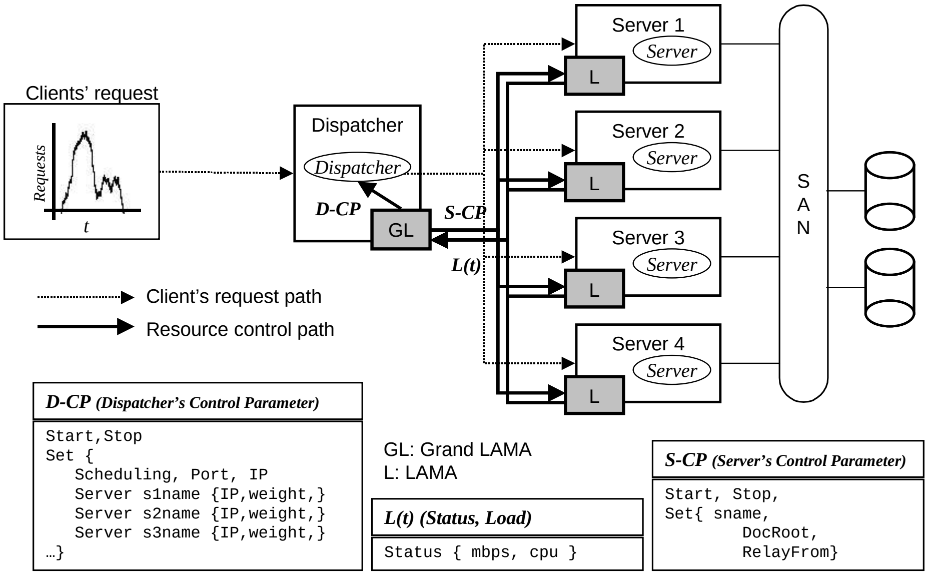
```
public interface Lama {

    public String status(String statusType)
    public boolean start()
    public boolean stop()
    public String set(String configXML)

} // Lama
```

Fig. 4. Lookup Service (LUS) for the management of a LAMA pool

discovered message only if the LAMA holds a correct key. Then, the LAMA can join the pool. The reallocation engine in the Grand LAMA can access a LAMA via RMI after looking up the LAMA. Also, the engine has to hold a correct key to invoke the LAMA methods. A malicious LAMA cannot discover the pool without a correct key, so that it cannot join the pool. The key should be distributed to a identified componenet, when the component is installed first time. JINI's *lease* interface makes the pool management robust to node failure. Leasing enables a LAMA to be listed in the pool for a given period of time; beyond that it has to renew its registration to avoid removal from the pool. JINI also provides a distributed security model, and through that the codes for the management can be distributed and executed in a secure way.

The operations of the lookup service (LUS) are shown in Figure 4. The LUS is originally from the reference implementation of JINI LUS, called REGGIE[10]. The LUS stores a set of *ServiceItem*s, which have IDs, a LAMA interface class, and attributes. A remote LAMA instantiates a *ServiceItem*, ITEM, and register it to the LUS. Then, the ID and lease duration are returned. Lookup also uses an instance of *ServiceItem*, ITEM, which specifies attributes of the needed LAMA. Attributes indicate the roles of a node; a web server, a dispatcher, or a streaming server. By specifying the attributes as a web server, a LAMA can be found, which is able to run a web server. If the attributes are not specified, all the *ServiceItem* corresponding to LAMAs will be returned.

Our CORE-Web server, which is managed by the Grand LAMA and ordinary LAMA, is shown in Figure 5. It includes a dispatcher, and back-end servers. The dispatcher distributes clients' requests over the backend servers, and then the back-end servers respond to the requests through accessing a SAN-based shared storage. The solid lines describe control paths between the Grand LAMA and LAMAs. The Grand LAMA collects the status (*L(t)*) of the back-end servers from LAMAs. *L(t)* contains node-wide status, like CPU utilization and network bandwidth. The Grand LAMA could reallo-



D-CP (Dispatcher's Control Parameter)

```
Start,Stop
Set {
   Scheduling, Port, IP
   Server s1name {IP,weight,}
   Server s2name {IP,weight,}
   Server s3name {IP,weight,}
…}
```

L(t) (Status, Load)

```
Status { mbps, cpu }
```

S-CP (Server's Control Parameter)

```
Start, Stop,
Set{ sname,
        DocRoot,
        RelayFrom}
```

Fig. 5. CORE-Web server and LAMA

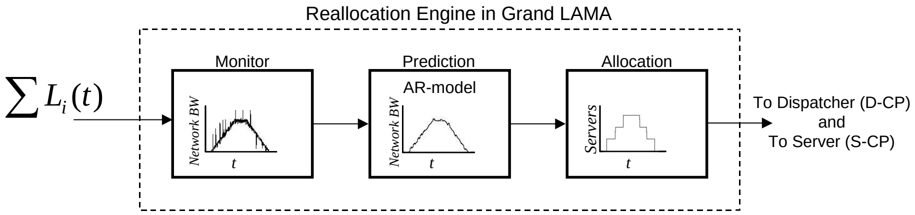


Fig. 6. Resource reallocation engine in the Grand LAMA

cate the nodes, by updating the control parameters of the dispatcher, D-CP, and those of servers, S-CP. Each parameter includes *start* and *stop* to initiate and destroy the server respectively. *Set* is for adjusting application specific attributes. D-CP has specific attributes such as a list of the back-end servers including IP addresses, host names, and weights for load-balancing. S-CP for a web server also has changeable attributes such as a hostname (sname), a root path of contents like HTML documents (DocRoot), and the address of an original source server (RelayFrom) in the case that the back-end servers relay a live media streaming.

The operation of a resource reallocation engine in the Grand LAMA is described in detail in Figure 6. The status of each node is gathered in the Grand LAMA. Therefore, the monitor module keeps overall resource usages at time *t*. The future resource demand is predicted using an autoregressive model. Also, the autoregressive model filters out noises in the signal of the resource usage. Based on the predicted resource demand, the allocation module adjusts the number of back-end servers in advance. This server reallocation is executed through updating D-CP and S-CP.

A goal of the resource reallocation is to save resources while meeting constraints on the application performance usually described in a Service Level Agreement (SLA). Therefore, a proper algorithm of the demand prediction should be deployed for the most cost-effective resource allocation.

A threshold-based heuristic algorithm[18] is simple but reactive to a sudden change in resource demands; however, it may cause unstable reallocation due to the high noise of input workloads. Also, it is not easy to determine the proper upper and lower thresholds.

A forecast-based algorithm is usually based on an autoregressive model, which predicts the future resource demands. It can capture long-term trends or cyclic changes like a time-of-day effect. Short-term forecasting may handle workload surges effectively, when the time overhead of reallocation is high [17]; however, inaccuracy of forecasting causes problems. We applied and compared both algorithms for the prediction in the reallocation engine, and will present the result below.

4 Prototype: Audio streaming service

Our prototype system provides a live audio streaming service. At the beginning, only one back-end node might be initiated as a streaming media server, such as icecast [19]. Upon detecting load increases, more nodes should be allocated for the service. An idle node could be chosen as an additional icecast server, and configured to relay the source streams from the first initiated icecast server. In this case, the server-side control parameter (S-CP) is *RelayFrom*, which describes the address of the original source server from which the audio stream is relayed. Each LAMA registers its interface to the pool (LUS). Then, the Grand LAMA composes a set of LAMAs dedicated to the audio streaming service, by looking up LAMAs from LUS. It constantly monitors its under-managed LAMAs to detect changing resource demands.

Resources can be measured with different metrics, according to the different aspects of the specific applications. Three major aspects of resources are acceptable, such as the CPU usage, network bandwidth, and disk storage. The performance of the application is highly correlated with these three metrics. SAR [20] produces many statistics about the system including the above metrics. Our LAMA measures resource utilization using the SAR, and then sends the status to Grand LAMA.

The capacity of the live audio streaming service depends solely on a network bandwidth, since it serves multiple users with only a single stream. When the source stream is igniting at 128 kbps, our single node could serve less than 750 concurrent connections reliably. With 750 concurrent connections, network utilization reached up-to 98%. With over 750 connections, the average streaming rates decrease rapidly, and the icecast server closes many connections, since server-side queues overflow due to the severe network congestion.

We compared two prediction algorithms. In a threshold-based prediction, we simply chose 90% as the upper threshold, and 80% as the lower threshold. The algorithm is described in Algorithm 1. Even though icecast server could spend 98% of the network

Algorithm 1 Reallocation algorithm for the threshold-based and AR-based prediction

```
$N_A \Leftarrow$ The Number of allocated servers (1)
$C \Leftarrow$ The network capacity of a node (100Mbps)
$t \Leftarrow$ The current time (0)
$p \Leftarrow$ Sampling period (2 seconds)
$L_i(t) \Leftarrow$ Network usage of a node $i$ at time $t$ in Mbps
$ut \Leftarrow$ Upper threshold (0.9)
$lt \Leftarrow$ Lower threshold (0.8)
$dcp$: Dispatcher's conrtrol parameters, a list of backend servers
$scp$: Server's conrtrol parameters, $RelayFrom$
loop
    $SUM_L \Leftarrow$ Sum of $L_i(t)$ for all $i$ {In case of AR, $SUM_L$ is a forecasted sum}
    $TC \Leftarrow N_A \cdot C$
    if $SUM_L > ut \cdot TC$ then
        Supplement an additional backend server $X$
        $N_A \Leftarrow N_A + 1$
        $dcp \Leftarrow dcp + X$
        $X.scp \Leftarrow$ The address of the original source server
    else if $SUM_L < lt \cdot (TC - C)$ then
        Release a node $V$
        $N_A \Leftarrow N_A - 1$
        $dcp \Leftarrow dcp - V$
    end if
    Sleep during $p$ periods
    $t \Leftarrow t + p$
end loop
```

bandwidth, we chose 90% as the upper threshold for reliable preparation, meaning that when the overall network utilization over the distributed back-end servers exceeds 90%, an additional back-end server is supplemented. When the utilization can be lower than 80% despite excepting one of back-end servers, the victim back-end server is released. When the resource utilization decreases, excess resources should be released or yielded to other service in order to waste. Excess resources should be released gradually to avoid a sacrifice of service qualities by abruptly closing innocent client's open connections. In this context, it is important to choose the lowest utilized victim for release. Otherwise, some kinds of connection migration techniques should be devised. In our experiment, we simply assumed that the rejected clients would request again by client-side programs, so we did not consider the service distinction of releasing resources.

Also, we implemented a forecast-based prediction algorithm using an autoregressive model, AR(1). Even though we can see a time-of-day effect in Figure 7, the workload is more autocorrelated with short-term history within 30 minutes than the one-day-before long-term one. We saw that long-term forecasting is much less accurate than short-term forecasting. The period (around 30 minutes) that shows reasonable forecasting accuracy, is enough to handle reallocation; therefore, we used a simple short-term (20 seconds ahead) forecasting of AR(1) model with a 40 seconds history. The reallo-

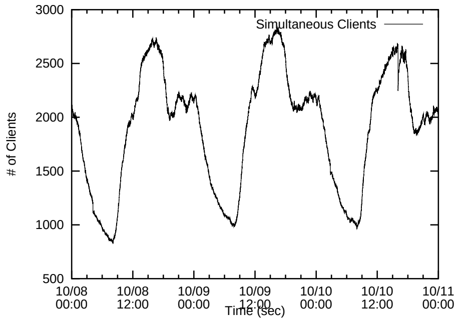


Fig. 7. Time-of-day effects in real traces from a popular audio streaming service

cation algorithm is similar to the threshold-based one, except using forecasted values rather than a simple sum of network usage.

5 Evaluation

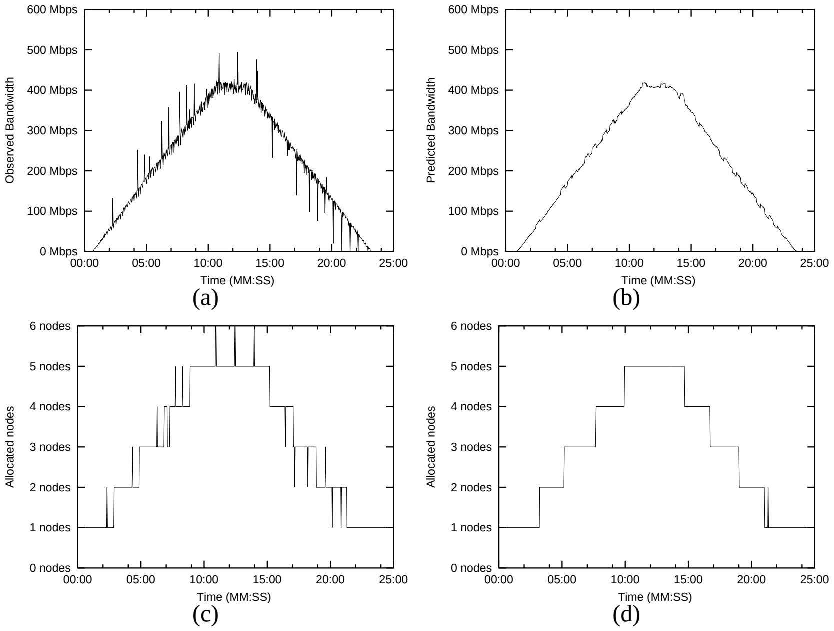
We have investigated real traces from a popular audio streaming service. The patterns of concurrent clients at October 9, 2003 are shown in Figure 7, and we can see the time-of-day effect. The number of listeners increased rapidly at the beginning of the day from 9 a.m.

We found that the steepest rate was 300 requests per minute, when we sampled traces every 10 seconds in October, 2003; therefore, we synthesized a workload that generates at the rate of 300 requests per minute for up to 3000 concurrent streams. The peak loads were sustained for 3 minutes, and then we removed clients' streams at 300 requests per minute as well as increasing rates. For a 128Kbps media stream, approximately 132Kbps bandwidth is required. The bandwidth includes client's TCP ACK and TCP/IP headers, so to follows the synthesized workload fully, more than 4 nodes are required, which are connected to 100Mbps network.

We observed that *discovery* takes quite a long but unpredictable time from 2 seconds to 20 seconds to discover the LAMA pool (LUS) since the JINI-based LAMA multicasts the discovery messages and waits for a few seconds until it gets discovery responses from the available LUS. However, after the discovery was completed, consequent communication did not produce latency. The discovery would be done only at the beginning, so that the unpredictable discovery time would not bother us.

The performance of the resource allocator is affected by monitoring intervals, and also by allocation overhead. In our Grand LAMA, a monitoring interval is 2 seconds. Allocation overhead was observed to be within 1 miniute.

The result of dynamic reallocation using threshold based reactive actions and prediction based proactive actions is shown in Figure 8. Since there are many noises in the monitored signal in the Figure 8-(a), the reactive reallocation shows resource cycling, in Figure 8-(c). On the other hands, the forecasted bandwidth usage in Figure 8-(d) seems to be filtered out. It shows a more stable resource reallocation than the reactive method.



(a) (b)

(c) (d)

Fig. 8. Dynamic reallocation: (a) Network bandwidth usage observed by the Grand LAMA, (b) Predicted network bandwidth using AR(1) model (c) Threshold-based reactive reallocation, (d) AR(1) model based proactive reallocation

6 Conclusion

We have proposed a framework for the autonomous management of large-scale clustered internet servers. Our autonomous management is based on distributed agents known as LAMAs. We adopt JINI technology for a flexible agent, which means that static configurations of networks are removed. LAMA utilized many features provided by JINI infrastructure in building a spontaneous network securely.

Our prototype system provided autonomous management for streaming media service. In order to adapt to the changing workload patterns, LAMA sent monitored statistics on each node. The Grand LAMA gathered them, inferred resource utilization, made decisions to demand or release resources.

The live audio streaming service is a simple example in which resources are represented only with network bandwidth. In the case of complex services including a web application server, overall statistics on resource utilization would be required. One challenging problem is inferring a system's capacity without prior knowledge or human intervention. For this, it is required to estimate application performance, which is a client's perceived quality of service in the case of internet service. Also, it is highly demanded to optimize resource allocation in a shared environment by multiple services since they would compete for resources.

References

1. Joo Young Hwang, Chul Woo Ahn, Se Jeong Park, and Kyu Ho Park: A Scalable Multi-Host RAID-5 with Parity Consistency, IEICE Transactions on Information and Systems, E85-D 7 (2002) 1086-1092
2. Seung Ho Lim, et. al.: Resource Volume Management for Shared File System in SAN Environment, Proceedings of 16 th International Conference on Parallel and Distributed Computing Sytems, Reno, USA August (2003)
3. Yang Hwan Cho, Chul Lee and Kyu Ho Park: Contents-based Web Dispatcher (CBWD), Technical Report, EECS, KAIST, January (2001)
4. Chul Lee and Kyu Ho Park: Kernel-level Implementation of Layer-7 Dispatcher (KID), Technical Report, EECS, KAIST, December (2002)
5. Sang Seok Lim, Chul Lee, Chang Kyu Lee, and Kyu Ho Park: An Advanced Admission Control Mechanism for a Cluser-based Web Server System, Proceedings of IPDPS Workshop on Internet Computing and E-Commerce (2002)
6. Chul Lee, Sang Seok Lim, Joo Young Hwang, and Kyu Ho Park: A Ticket based Admission Controller(TBAC) for Users' Fairness of Web Server, Proceedings of 3rd Interneational Conference on Internet Computing (2002)
7. Redhat: Linux Advanced Server, http://www.redhat.com/
8. Jeffrey O. Kephart and David M. Chess: The Vision of Autonomic Computing, IEEE Computer, January (2003)
9. M. Parashar: AutoMate: Enabling Autonomic Applications, Technical Report Rutgers University, November (2003)
10. Sun Microsystems: JINI Network Technology, http://wwws.sun.com/software/jini/
11. PXES: Linux thin client project, http://pxes.sourceforge.net/
12. Etherboot: Remote netowrk boot project, http://www.etherboot.org/
13. Paul Anderson: LCFG A large-scale UNIX configuration system, http://www.lcfg.org/
14. Redhat: Kickstart, http://www.redhat.com/
15. Y. Diao, J. L. Hellerstein, S. Parekh, J. P. Bigus: Managing Web server performance with AutoTune agents, IBM Systems Journal Vol 42, No 1 136-149 (2003)
16. Paul Anderson and Patric Goldsack and Jim Paterson: SmartFrog meets LCFG: Autonomous Reconfiguration with Central Policy Control Proceedings of LISA XVII USENIX San Diego, USA (2003)
17. E. Lassettre, et.al.: Dynamic Surge Protection: An Approach to Handling Unexpected Workload Surges with Resource Actions that Have Lead Times, , Proceedings of 14th IFIP/IEEE Disitributed Systems: Operations and Management, (2003)
18. Abhishek Chandra, Weibo Gong, and Prashant J. Shenoy: Dynamic Resource Allocation for Shared Data Centers Using Online Measurements, Proceedings of SIGMETRICS (2003)
19. icecast streaming media server: http://www.icecast.org
20. System Activity Reporter: http://perso.wanadoo.r/sebastien.godard